VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Datové struktury pro indexování tabulky trojic

Data Structures for Indexing Triple Table

2016 Denis Melišek

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání bakalářské práce

Student: **Denis Mališek**

Studijní program: B2647 Informační a komunikační technologie

Studijní obor: 2612R025 Informatika a výpočetní technika

Téma: Datové struktury pro indexování tabulky trojic
Data Structures for Indexing Triple Table

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je implementace datových struktur AVL-strom a Grid File pro indexování tabulky trojic RDF. Student dále srovná implementované struktury s ostatními strukturami využívanými k indexování sémantických dat. Datové struktury budou implementovány v C++.

1. Proveďte rešerši datových struktur využívaných v sémantických databázích.
2. Implementujete datové struktury AVL-strom a Grid File.
3. Proveďte experimenty a zhodnoťte implementované datové struktury.
4. Srovnejte výkon implementovaných struktur s dalšími strukturami pro indexovaní tabulky trojic.

Seznam doporučené odborné literatury:

[1] ADELSON-VELSKII, G. M.; LANDIS, Evgenii Mikhailovich. An information organization algorithm. In: Doklady Akademia Nauk SSSR. 1962. p. 263-266
[2] NIEVERGELT, Jürg; HINTERBERGER, Hans; SEVCIK, Kenneth C. The grid file: An adaptable, symmetric multikey file structure. ACM Transactions on Database Systems (TODS), 1984, 9.1: 38-71
[3] KALEV, Danny; SCHMULLER, Joseph. The ANSI/ISO C++ professional programmer's handbook. Que Corp., 1999

Formální náležitosti a rozsah bakalářské práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí bakalářské práce: **Ing. Roman Meca**

Datum zadání: 01.09.2015
Datum odevzdání: 29.04.2016

doc. Dr. Ing. Eduard Sojka
vedoucí katedry

prof. RNDr. Václav Snášel, CSc.
děkan fakulty

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 29. dubna 2016 .....

Souhlasím se zveřejněním této bakalářské práce dle požadavků čl. 26, odst. 9 Studijního a zkušebního řádu pro studium v bakalářských programech VŠB-TU Ostrava.

V Ostravě 29. dubna 2016 .....

Rád bych poděkoval panu Ing. Romanovi Mecovi za vedení mé práce a za velkou snahu, podporu, trpělivost a laskavost při vypracovávání bakalářské práce.

Abstrakt

Bakalářská práce je zaměřená na datové struktury využívané v sémantických databázích. Rešeršní část práce popisuje základní datové struktury, které se používají k indexování dat. Implementační část popisuje dvě datové struktury, a to konkrétně AVL-strom a Grid File, analyzuje jejich struktury i funkcionality a následně znázorňuje, jak vypadá jejich zavádění. Experimentální část testuje obě implementované datové struktury a porovnává je s ostatními strukturami tak, že provádí sadu definovaných příkazů a měří časy provedení dotazů. Závěrečná část práce analyzuje a srovnává naměřené časy pro všechny datové struktury a vyhodnocuje jejich výsledky.

Klíčová slova: indexovací struktura, sémantická databáze, RDF, SPARQL, B-strom, R-strom, AVL-strom, Grid File, datová struktura, bitmapa, index

Abstract

Bachelor thesis is focused on the data structures used in the semantic databases. The research part describes basic data structure that are used for data indexation. The implementation part describes two implemented data structures, AVL-tree and Grid File, analyses their structure and functionality and visualizes their implementation. The experimental part tests implemented data structures and compares them with the others by executing set of defined orders and by measuring time needed for requests accomplishment. The final part of the thesis analyses and compares measured times for all data structures and evaluates the results.

Key Words: index structure, semantic database, RDF, SPARQL, B-tree, R-tree, AVL-tree, Grid File, data structure, bitmap, index

Obsah

Seznam použitých zkratek a symbolů

RDF	–	Resource Description Framework
SPARQL	–	SPARQL Protocol and RDF Query Language
SQL	–	Structured Query Language
XML	–	Extensible Markup Language
AVL-tree	–	Adelson-Velsky and Landis tree
URI	–	Uniform Resource Identifier

Seznam obrázků

Seznam tabulek

1 Úvod

V dnešní době už známe mnoho datových struktur od těch nejjednodušších jako: pole, list a další, po ty složitější jako například: stromy, grafy nebo hashe. A právě ty složitější struktury můžeme vidět v databázích, kde se nejvíce používají právě zmiňované stromy a to především: b-strom a jeho varianty. Ty struktury mají především význam pří efektivním třídění a vyhledávání dat.

Datové struktury se používají v databázích jako základní konstrukce, která slouží ke správě dat. Data v databázích se mohou lišit svým typem, například v relačních databázích jsou data uloženy jako řádky v tabulce, v naopak v sémantické databáze pracují tzv. trojicemi a mohou být spojovány se sémantickými weby, které mohou složit jako zdroje dat.

Právě tyto struktury používají indexy k práci s daty. To může výrazně zrychlit operace prováděné nad danou strukturou, při použití indexů může například přistupovat hned k datům na daném indexu nebo rovnou indexovat celé stránky.

Každá databáze potřebuje ke své činnosti datové struktury, ať už se jedná o relační, sémantickou, XML nebo jinou databázi, rozdíl je jen v tom, s jakými daty pracují. Datové struktury neboli též nazývané indexovací struktury se používají na různých místech, a jedno z nich jsou i databáze.

Datové struktury používající v sémantických databázích pracují s daty, jako jsou například: trojice (s, p, o). Proto se takové struktury musejí přizpůsobit na práci s takovými daty. Sémantické databáze si neukládají svá data do relačních tabulek, ale namísto toho zpracovávají tzv. RDF data. Jedná se o datový model, který popisuje informace přijatelnějším způsobem pro stroj a navíc reprezentuje meta-data (data o datech). Obecně se jedná o nějaký zdrojový dokument, který je čitelný jak pro člověka, tak i pro stroj. Zároveň je to i grafový formát, protože svá veškerá data může znázornit pomocí grafu s orientovanými hranami, jenž pak můžeme zapisovat jako množinu trojic.

Pro RDF existuje jazyk SPARQL, něco jako SQL pro relační databázi. Jedná se o sémantický dotazovací jazyk, který pro své dotazování využívá výše zmiňovaná RDF data, kde jako základní informační prvek je trojice, která se skládá, jak se dále dozvíme z podmětu, vlastnosti a předmětu.

Mým cílem pak bude otestovat takové dotazy na různými datovými strukturami a naměřit jejich časy, které se budou z největší pravděpodobnosti lišit. Provedu testování a následně budu moci porovnat a vydedukovat, jaká struktura tento problém zpracovává nejlépe.

2 Základní pojmy

2.1 URI (Uniform Resource Identifier)

V překladu by to mohlo být asi takto: jednotný identifikátor zdroje. URI je jednoduše webový identifikátor, neboli řetězec začínající „http:“ nebo „ftp:“, které často najdeme v internetovém prohlížeči jako řetězec představující internetovou stránku. Odkazuje na fyzické zdroje stránky a specifikuje zdroje informací. Kdokoliv může vytvořit URI, a její vlastnictví jasně zastoupené, takže tvoří ideální základní technologii, se kterou se dá vytvořit globální web. Cokoliv, co má URI, je považováno, že je „na webu“. Syntaxe URIs je důsledně řízené komunitou IETF, která publikovala „RFC 2396“ jako obecnou URI specifikaci. [1]

2.2 RDF (Resource Description Framework)

Trojice může být jednoduše popsána jako tři URI. Jazyk, který využívá tři URI v jeho způsobu, se nazývá RDF. W3C vyvinul XML serializaci RDF. RDF XML je považován k tomu, aby byl standartní, zaměnitelný formát pro RDF na sémantickém webu., ačkoliv to není jenom formát. Pro příklad: „Notation3“ je výborná alternativní serializaci prostého textu. Tady je příklad jednoduché trojice tří URI trojic:

<http://rdf.org/#x> <http://rdf.org/#y> <http://rdf.org/#z>

Když informace je v RDF formě, stane se lehkou k její zpracování, protože RDF je generický formát, který už má mnoho nástrojů na její syntaktické rozebrání. XML RDF je celkem mnohomluvná specifikace a může brát některé zvyklosti (například učit se XML RDF řádně, potřebuješ rozumět trochu o XML a jmenných prostorech atd.), ale můžeme se podívat na příklad XML RDF:

```
<rdf:RDF
    xmlns = "http://kniha/instace/"
    xmlns:v = "http://kniha/slovnik/"
    xmlns:rdf = "http://www.knihy.cz/1993/2/11-kniha-o-rdf-ns#">
    <rdf:Description rdf:about = "http://kniha/instace/kniha1">
        <rdf:type rdf:resource = http://kniha/slovnik/kniha">
        <v:author>Pepa Smolik</v:author>
        <v:title>RDF modely</v:title>
    </rdf:Description>
</rdf:RDF>
```

Obrázek 1: Ukázka kódu XML RDF

Tento kousek RDF jednoduše říká, že kniha má název „RDF modely“ a byla napsána někým, kdo se jmenuje „Pepa Smolik“. [1]

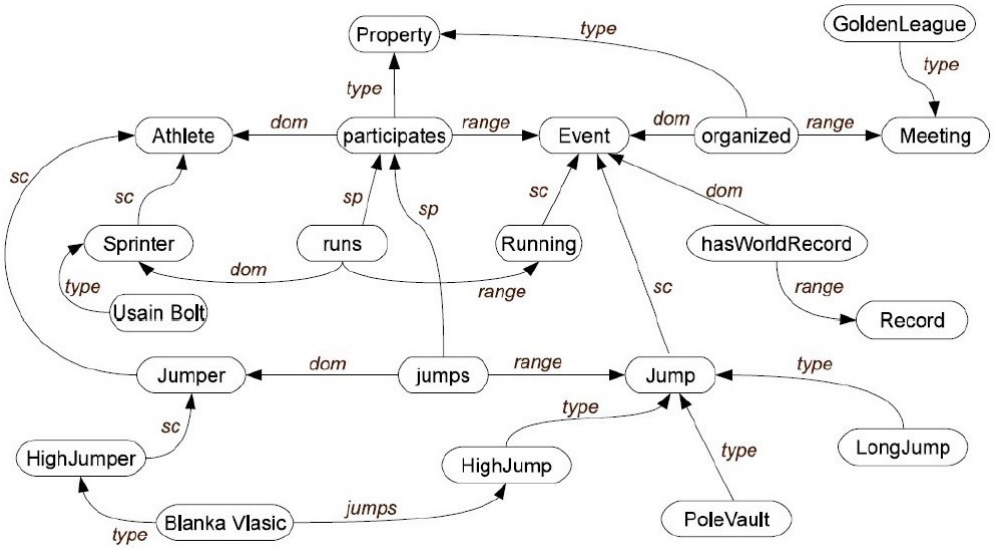
Definice 1 (RDF trojice) *Předpokládejme, že existují párované, disjunktní a nekonečně kolekce I, B a L, kde I reprezentuje kolekci „IRIs“ (znárodněných identifikátorů zdrojů), B je*

kolekce prázdných uzlů a L je kolekce literálů. Nazýváme trojici $(s, p, o) \in (I \cup B)(I \cup B \cup L)$ *RDF trojicí, kde s reprezentuje podmět či subjekt (angl. subject), p reprezentuje predikát (angl. predicate) a o jako „object“ neboli nějaký předmět RDF trojice. [3]*

Definice 2 (Tabulka trojic) *Tabulka trojic je kolekce RDF trojic, tzn. je to reprezentace RDF grafu. V tabulce 1 vidíme fragment tabulky trojic z RDF grafu na obrázku 3. Uložiště trojic nebo RDF databáze jsou nástroj umožňující skladovat RDF graf a efektivní zpracování dotazů. Avšak obvykle požadujeme i další operace jako obnovení, vložení nebo vymazání. [3]*

Tabulka 1: Fragment RDF tabulky trojic [4]

Subject	Property	Object
LongJump	Type	Jump
Blanka Vlasic	Jumps	HighJump
GoldenLeague	Type	Meeting



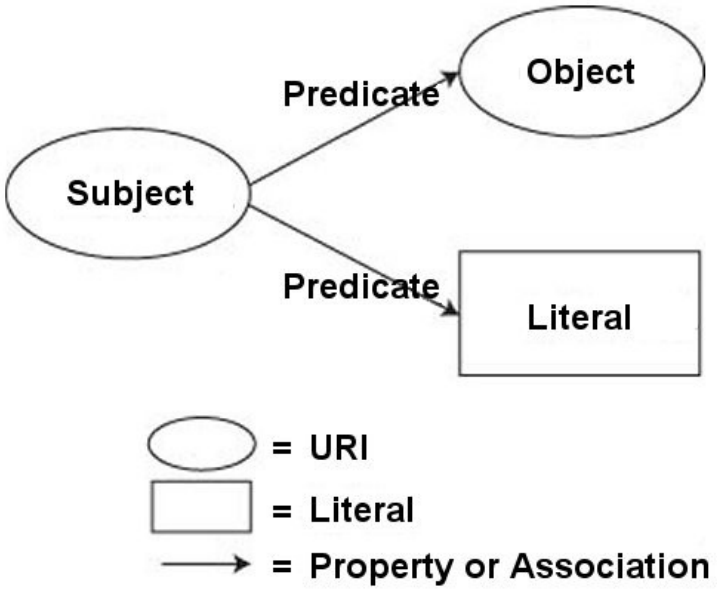
Obrázek 2: Příklad RDF grafu [4]

2.3 SPARQL

Jako SQL je nejdůležitější dotazovací jazyk pro relační databáze, tak SPARQL je nejdůležitější dotazovací jazyk pro sémantický web. Dnes je SPARQL uznán jako standartní dotazovací jazyk a je podporován hlavní databázovým prodejcem jako například Oracle.

SPARQL nabízí mocné možnosti k dotazování RDF trojic a grafů a podporuje řadu dotazovacích schopností. Výsledek SPARQL dotazů může být řízen, limitován a kompenzován příslušným počtem elementů.

Hlavní komponentou SPARQL dotazů je sada vzorů trojic s p o. Trojice s p o odpovídá k: „subject“ (s) – nějaký podmět, „predicate“ (p) – většinou nějaká vlastnost, a „object“ (o) – nějaký předmět RDF trojice, ale mohou to být proměnné stejně jako RDF výraz. Během SPARQL dotazu uživatel specifikuje známý RDF výraz trojic a zanechá neznámou proměnou ve vzorci trojic. Ta samá proměnná se může objevit v několika vzorcích trojic a tedy implikovat spojení. Vzorec trojice se porovnává podmnožinu RDF dat, kde RDF výraz ve vzorci trojice odpovídá těm v RDF datech. [3]



Obrázek 3: Vyobrazení trojice RDF [6]

Základní dotazovací konstrukce SELECT dotazu zahrnuje SELECT <projekce> WHERE <sekvence vyhledávacích vzoru pro vyhledání trojic>. Neznámá proměnná v SPARQL se definuje symbolem „?“ a jméno reprezentuje hlavní odlišnost v porovnání SQL. Definují neznámé hodnoty o, s nebo p ve vzoru rovněž jako vztah mezi vzory trojic. Odlišujeme čtyři typy SPARQL dotazů:

1. SELECT – vrací výslednou relaci definovanou projekcí a vzory
2. ASK – podobný k SELECT dotazu, avšak vrací booleovskou hodnotu, neboli vrací „pravda“, jestli výsledek není prázdný, jinak nepravda
3. CONSTRUCT – dovoluje formátovat vlastní výsledný graf nad vrácenými trojicemi dle vzorů
4. DESCRIBE – vrací uzel (a jeho sousedy) definované dle vzorů

Forma <vzor> určuje selektivitu dotazu nad tabulkou trojic. Můžeme odlišit „bodový“ dotaz (s,p,o), který vrací žádnou nebo jednu trojici od rozsahového dotazu, kde dotaz (s,*,*) může vracet více trojic než dotaz (s,p,*).
Příklad 1 (SPARQL dotazy):

1. SELECT ?s ?p ?o WHERE ?s ?p ?o
   Tenhle dotaz vrací celou tabulku trojic a je reprezentován rozsahovým dotazem (*,*,*).

2. ASK <Blanka Vlasic> <jumps> <HighJump>
   ASK dotaz vrací "pravda"v případě, že trojice existuje v grafu. Tento dotaz reprezentuje „bodový“ (s,p,o) dotaz nad tabulkou trojic. [5]

3 Datové struktury v sémantických databázích

Jedná se o nějakou množinu dat, která slouží k uchovávání dat. Data v ní jdou samozřejmě i vkládat, uspořádat či smazat. Taktéž je důležité vyhledávání, které závisí na daném typu struktury a jejím možnostem. Pokud se dá velikost struktury měnit, říkáme, že se jedná o dynamický typ. Můžou být lineární (fronta, pole, zásobník) nebo nelineární (stromy). Aspekty pro výběr datové struktury mohou být různě jako třeba: čas potřebný k provedení operace s daty, vyhledávání dat nebo použitá paměť. V mém případě budeme hledat takové struktury, které se mohou použít pro skladování trojic dat a navíc se požívají v sémantických databázích.

3.1 B-strom

3.1.1 Historie

Počátky B-stromu šahají ke konci 60. let 20. století. Začalo to v publikaci R. Bayera a E. McCreighta: „Organization and maintenance of large ordered indexes“, kde se zabývají problémem organizace a zachování indexu pro dynamicky se měnící přístupového souboru. Ve snaze vyvinout algoritmus pro skladování a získávání dat z počítače, se pak publikovala novější verze se jménem: „Mathematical and Information Sciences Report No. 20“ v roce 1970.

Od té doby pak mnoho týmu pracovalo na zlepšení základní myšlence B-stromu, což vedlo k vymyšlení mnoho různých variací B-stromu jako například: B+-stromu, B*-strom, UB-strom, B-strom-P a další. [7]

3.1.2 Základy

Jedná se o binární vyhledávací strom, který může složit mnoha k účelům jako například ke skladování dat. Dokonce je to i sebe vyvažující datová struktura, která svá data třídí a dovoluje vyhledávání pomocí sekvenčního přístupu, jelikož jsou data ve stromě setříděna většinou dle klíče, jenž každý uzel obsahuje. A následně dovoluje i vložení a smazání uzlů ze stromu a to vše v logaritmickém čase.

Jelikož tato struktura spadá pod binární vyhledávací stromy, které se skládají z uzlů, proto každý uzel může mít více než dva potomky právě kvůli své sebe vyváženosti právě proto, když nastane situace, že by se mohl utvořit strom, jenž by měl jen pravý podstrom, proto se strom upraví tak, aby měl co možno nejmenší výšku, a proto může kořen nebo další uzly ukazovat na více uzlů než jen na dva (také záleží jakého je řádu), jak to bývá u binárních stromů.
Existují i varianty B-stromu:

1. **B+-strom** – v této variantě strom skladuje kopie klíčů ve svých vnitřních uzlech, pomocí kterých se dostaneme na listové uzly, ve kterých skladuje dané klíče, data nebo ukazatele na data, a zároveň v listových uzlech skladuje ukazatele na vedlejší uzly.

2. **B*-strom** – definuje, že v každý uzel je nejvýše zaplněn do 2/3 (místo jenom do 1/2). Vložení zaměstnává lokální redistribuční schéma na oddálení rozdělení, dokud dva příbuzné uzly nejsou plné, pak se dva uzly rozdělí do tří, jenž každý je plný do 2/3. Toto schéma zaručuje využití nejméně 66%.

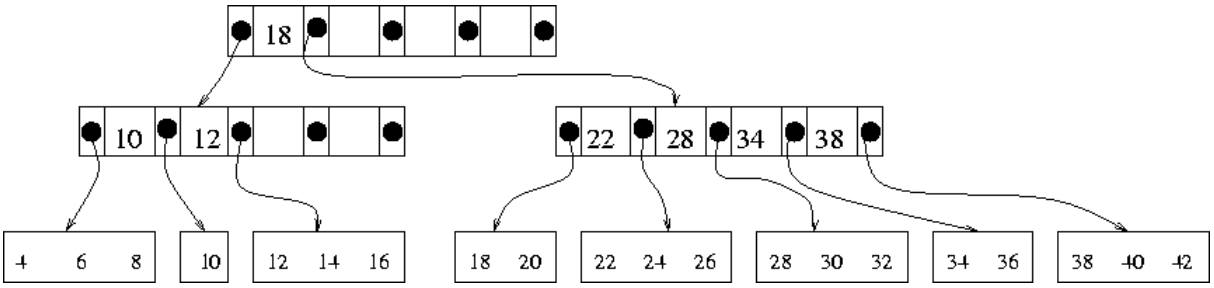
3. **B-strom-P** – vykonává binární vyhledávání uvnitř každého uzlu, což někdy i jednoduché lineární vyhledávání nebo další technika může nabídnout lepší výkon.

4. **UB-strom** – navržen pro skladování a efektivní získávání vícerozměrných dat.

Navíc se jedná o velmi efektivní strukturu pro vyhledání a uchování hodnot, pokud se nevejdou do operační pamětí a musí být uloženy jinde, třeba na pevném disku. Jeho složitost vyhledávání je v nejhorším případě $log_m(M)$, kde M je počet záznamů ve stromě. B z názvu nikdy nebylo autory vysvětleno, ale většinou se předpokládá, že by to mohlo být spojeno s příjmením Bayer, anebo by to mohlo znamenat „Balance“ v překladu vyvážený. [8]

Definice 3 (B-strom) *B-strom řádu m je m-ární vyhledávací strom s následujícími vlastnostmi:*

1. *Všechny listové uzly (uzly, které nemají žádného potomka) jsou na stejné úrovni stromu.*
2. *Všechny vnitřní uzly mají mezi $\frac{m}{2}$ a m potomků (nemusí platit pro kořen).*
3. *Každý uzel má nejvýše m+1 klíčů, kromě kořene.*
4. *Každá cesta z kořene do listového uzlu má stejnou délku.*
5. *Kořen má maximálně m potomků.*



Obrázek 4: Ukázka B-stromu řádu 5, ve kterém jsou nejvíce 3 záznamy [9]

3.1.3 Vyhledávání

Vyhledat prvek v B-stromu je celkem lehké. Začneme od kořene, každý uzel obsahuje klíč, který budeme porovnávat s hledanou hodnotou. Když klíč je menší než hledaná hodnota, tak se podíváme do ukazatele na pravého potomka, tedy přejdeme do pravého potomka, pokud klíč je větší

než hledaná hodnota, přejdeme do levého potomka, takhle pokračujeme, dokud nedojdeme do listového uzlu, který už nemá žádné potomky nebo se klíč shoduje s hledanou hodnotou, a tedy jsme nalezli to, co jsme chtěli. Pokud dojdeme do listového uzlu a ani po té se klíč neshoduje s hledanou hodnotou, pak jsme nic nenalezli a skončí vyhledávání.

3.1.4 Vkládání

Vkládání je opět celkem jednoduché, pomoci stejného algoritmu jako při vyhledávání se nalezne příslušný uzel, kam tu danou hodnotu vložit a pokud uzel ještě není plný, tak se do něj vloží prvek a není co řešit, ale pokud je uzel skoro plný, myšleno, že obsahuje n – 1 prvků, kde n je maximální počet prvků v uzlu, tak bude nutné rozdělit uzel a to tak, že se vezme prostřední prvek z uzlu, který se rozdělí a vloží se do svého rodiče a následně se levé prvky zůstávají v původním uzlu a dále se vytvoří nový uzel a tam se vloží pravé prvky neboli větší prvky než ten zmiňovaný prostřední prvek, který se přesunul do rodiče. Pokud i rodič po přidání toho prostředního prvku bude plný, stane se to samé a v nejhorším případě to může dojít až ke kořeni a kdyby se i on rozdělil, tak by se strom zvětšil ve své výšce o jedna.

3.1.5 Mazání

Při mazání se můžeme setkat se dvěma případy. V prvním mažeme prvek z listového uzlu, a tedy ho můžeme bez problému smazat a je to hotové, jenom pokud by pak ten uzel nesplňoval definici B-stromu, tak bychom museli provést pár změn jako třeba: spojit s jiným uzlem, nebo přidat prvek z jiného sourozeneckého uzlu. V druhém případě mažeme z nelistového uzlu, tam si pak musíme dát pozor, abychom neporušili strukturu stromu. Klíč nejprve nalezneme a poté ho smažeme a na jeho místo vložíme jeho prvního následníka. Pokud by opět byl nedostatečný počet prvku po jeho smazání v uzlu, tak bychom to museli napravit třeba spojením uzlu, nebo přidáním jiných prvku se sousedních uzlu tak, abychom neporušili pravidla B-stromu. [10]

3.2 R-strom

3.2.1 Historie

R-strom navrhl A. Guttman v roce 1984. Klíčová myšlenka byla seskupit blízké objekty a reprezentovat je s jejími tzn. „minimálními ohraňujícími obdélníky“ v další vyšší úrovni stromu. „R“ v názvu R-strom znamená obdélník (Rectangle). [11]

3.2.2 Základy

R-strom je výškově vyvážený strom podobný B-stromu s indexováním záznamů v jeho listových uzlech obsahující ukazatele na datové objekty. Uzly odpovídají diskovým stránkám, jestliže index je disku odolný a struktura je tak navržena, pak prostorové vyhledávání vyžaduje navštívení

pouze malého množství uzlů. Index je zcela dynamický, vložení a vymazání muže být spojeno s vyhledáváním bez žádné vyžadující reorganizaci. [11]

Tato struktura se používá pro prostorové přístupové metody, například pro indexování vícerozměrných informací jako: geografické souřadnice, obdélníky nebo polygony. [11]

3.2.3 R-strom jako indexovací struktura

Prostorové databáze se skládají z kolekce mnoha „tuple“ (konečný uspořádaný list elementů) reprezentující prostorové objekty a každý „tuple“ má unikátní identifikátor, pomocí kterého může být získán. Listové uzly obsahují indexové záznamové vstupy ve formě (I, „tuple identifikátor“), kde identifikátor se odkazuje na „tuple“ v databázi a I je n-rozměrný obdélník, který je jakoby ohraňující prostor prostorového objektu indexovaného I = (I_0,I_1, ..., I_{n-1}), kde „n“ je počet dimenzí. Nelistové uzly obsahují vstupy ve formě (I, „ukazatel na potomka“), kde ten ukazatel je adresa nižšího uzlu v R-stromě a I zahrnuje všechny obdélníky v nižších uzlových vstupech.

Dejme tomu, že M je maximální počet vstupů, které budou sedět v jednom uzlu a m <= $\frac{m}{2}$ bude parametr specifikující minimální počet vstupů v uzlu, pak R-strom má následující vlastnosti:

1. Každý listový uzel obsahuje mezi m a M indexových záznamů, pokud to není kořen
2. Pro každý indexovací záznamový (I, „tuple identifikátor“) v listovém uzlu, I je nejmenší obdélník, který prostorově obsahuje n-rozměrové datové objekty reprezentující indikujícím „tuplem“.
3. Každý nelistový uzel má mezi m a M potomků, pokud se nejedná o kořen.
4. Pro každý vstup (I, „ukazatel na potomka“) v nelistovém uzlu, I je nejmenší obdélník, který prostorově obsahuje obdélníky v potomkově uzlu.
5. Kořenový uzel má nejméně dva potomky, pokud není listový uzel.
6. Všechny listy jsou na stejné úrovní stromu. [11]

3.2.4 Dodatek k velikost R-stromu

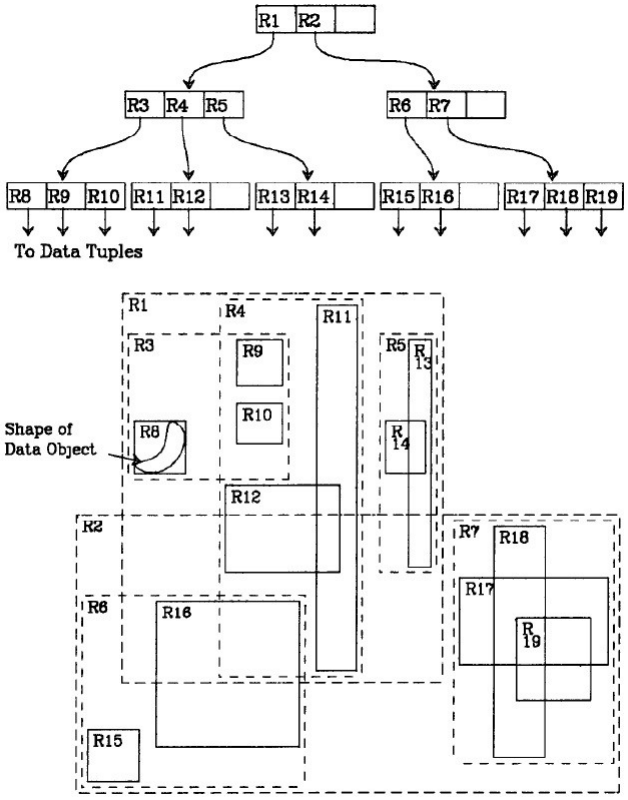
R-strom, který obsahuje N záznamů, pak jeho výška je nejvýše $|log_m(M)|$ - 1, poněvadž počet větví v každém uzlu je minimálně m a maximálně $|\frac{N}{m}| + |\frac{N}{m^2}| + \ldots + 1$. $\frac{m}{M}$ je potom nejhorší prostorové využití R-stromu.

3.2.5 Vyhledávání

V následném popisu je obdélník indexového záznamu E písmeny I_E a id je označeno p_E.
Algoritmus Search: najde všechny indexové záznamy, které překrývají S.

1. [Prohledej podstromy] Pokud T není list, prohledej všechny záznamy E a zjisti, jestli S se překrývá s I_E. Pro všechny takové E zavolej Search se stromem, který je zakořeněný v p_E.

2. [Prohledej list] Pokud T je list, procházej všechny záznamy E a zjisti, jestli se S překrývá s I_E. Pokud ano, E je výsledný záznam.



Obrázek 5: Jednoduchá ukázka R-stromu pro 2D obdélníky [11]

3.2.6 Vkládání

Vložení nových záznamů je podobné jako v případě B-stromu v tom, že nové záznamy se přidají do listů, pak uzly, které jsou plné, se rozdělí a rozdělení uzlu se síří stromem až k jeho vrcholu. Algoritmus Insert – vkládá nový záznam E do stromu T.

1. [Najdi pozici] Zavolej proceduru Choose-Leaf pro výběr uzlu L, do kterého se umístí E.

2. [Přidej záznam do listového uzlu] Pokud L má místo pro záznam, vlož tam E. Jinak zavolej Split-Node a získej L a LL obsahující E a všechny staré záznamy z L.

3. [Rozšiř změny nahoru] Zavolej proceduru Adjust-Tree na L (i s LL, pokud bylo provedeno rozdělení).

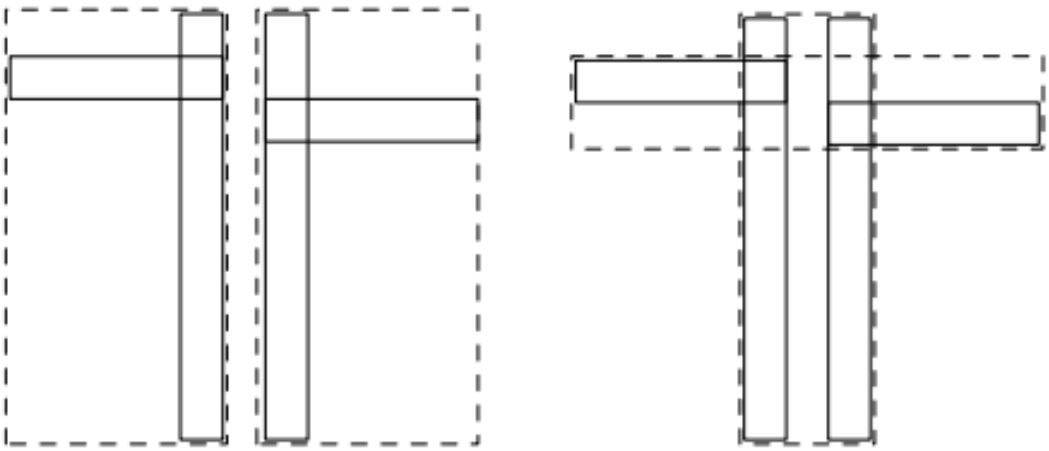
4. [Zvyš výšku stromu] Pokud se rozdělení uzlu postupně dostalo až ke kořeni, vytvoř nový kořen, jehož potomci budou výsledné dva uzly (L a LL).

3.2.7 Mazání

Algoritmus pro smazání záznamu E z R-stromu:

1. [Najdi uzel, který obsahuje E] Vyvolej Find-Leaf a najdi tak listový uzel L obsahující E. Pokud takový uzel neexistuje, skonči.
2. [Smaž záznam] Odstraň E z L.
3. [Šiř změnu] Vyvolej Condense a předej jí L.
4. [Zmenši výšku stromu] Pokud kořen má jen jednoho potomka po předchozím kroku, udělej z toho potomka nový kořen.

Detaily algoritmu jako například: FindLeaf či Choose-List použitých v předchozích algoritmech pro vkládání nebo mazání jsou více popsány v [11].



Obrázek 6: Vlevo je špatné rozdělení a vpravo je správné rozdělení [11]

3.3 Bitmapa

Ve výpočetní technice, bitmapa je mapování z nějaké oblasti (například z rozsahu celých čísel) na bity, což jsou hodnoty představující jedničku nebo nulu. Je taky nazývaná jako bitové pole či bitmapový index.

Bitmapa je typ paměťové organizace nebo souborový formát obrázku používaný pro skladování digitálních fotek. Termín „bitmapa“ pochází z terminologie počítačového programování, myšleno jako mapa bitů nebo prostorově mapované pole bitů. [12]

3.3.1 Bitmapový index

Bitmapový index je speciální typ struktury používaný databázovými systémy k optimalizaci vyhledávání a získávání nízko-proměnlivých dat. Logická perspektiva bitmapového indexu je přemýšlet o nich jako mřížka, v čem sloupce reprezentují každou diskrétní datovou hodnotu a řádky reprezentují pozici každého řádku v tabulce. Každá buňka bude obsahovat buď 1 nebo 0, pravda nebo nepravda.[13]

Základní bitmapový index je typicky používán k indexaci hodnoty jednoho sloupce X v tabulce. Tento index se skládá z nařízené sekvence klíčových hodnot reprezentující odlišné hodnoty sloupce a každá klíčová hodnota je sdružena s bitmapou, která specifikuje kolekci řádků v tabulce pro který sloupec X má danou hodnotu. Bitmapa má tolik bitů jako počet řádků v tabulce a k-tý řádek je si rovný s klíčovou hodnotou asociovanou s bitmapou a 0 pro jakýkoliv další sloupcovou hodnotu. Tabulka 2 ukazuje základní bitmapový index na tabulce s devíti řádky, kde sloupce X, jenž je indexovaný, má celočíselný rozsah od 0 do 3. Říkáme, že sloupcová kardinalita X je 4, protože má 4 odlišné hodnoty. Bitmapový index pro X obsahuje 4 bitmapy ukázané jako $B_0, B_1, \ldots, B_3$ s dolními indexy odpovídající k hodnotě, kterou reprezentují. V tabulce 2 druhý bit sloupce B_1 je 1, protože druhý řádek sloupce X má hodnotu 1, zatímco odpovídající bity sloupců $B_0, B_2 a B_3$ jsou všechny 0. [14]

Tabulka 2: Bitmapový index pro sloupce X. Sloupce B_0 – B_3 jsou bitmapy. [14]

RID	X	B_0	B_1	B_2	B_3
0	2	0	0	1	0
1	1	0	1	0	0
2	3	0	0	0	1
3	0	1	0	0	0
4	3	0	0	0	1
5	1	0	1	0	0
6	0	1	0	0	0
7	0	1	0	0	0
8	2	0	0	1	0

4 Teorie a implementace struktur Grid File a AVL stromu

Tahle část se bude věnovat dvěma datovým strukturám, které jsou úkolem implementace, a následného testování. Jedná se, dalo by se i říct, o staré struktury, které byly vymyšleny a popsány velmi dávno a měli představovat způsob, jak skladovat data. První struktura je Grid File, specifická tím, že je přizpůsobená pro vícerozměrná data, v našem případě se bude jednat o trojrozměrná data (trojice). Druhá struktura je AVL-strom a tato datová struktura se vyznačuje především tím, že se jedná o vyváženou strukturu a to, co to znamená, si osvětlíme později.

Při implementaci datových struktur budeme používat programovací jazyk C++ a jako vývojové prostředí jsme si vybrali Microsoft Visual Studio 2013, ve kterém budou následně probíhat experimenty a testování daných struktur.

4.1 Grid File

Když se řekne Grid File, tak z největší pravděpodobností to nikomu nebude nic říkat a to může být zapříčiněno tím, že se jedná o velmi starou strukturu (80. léta minulého století), která byla navržena v době, když se ještě nepředpokládalo, že budeme mít k dispozici tolik paměti jako dnes, myšleno jednotky GB v operační paměti či u pevných disků dokonce TB. Proto byla navržena, aby šetřila místo.

4.1.1 Grid File jako datová struktura

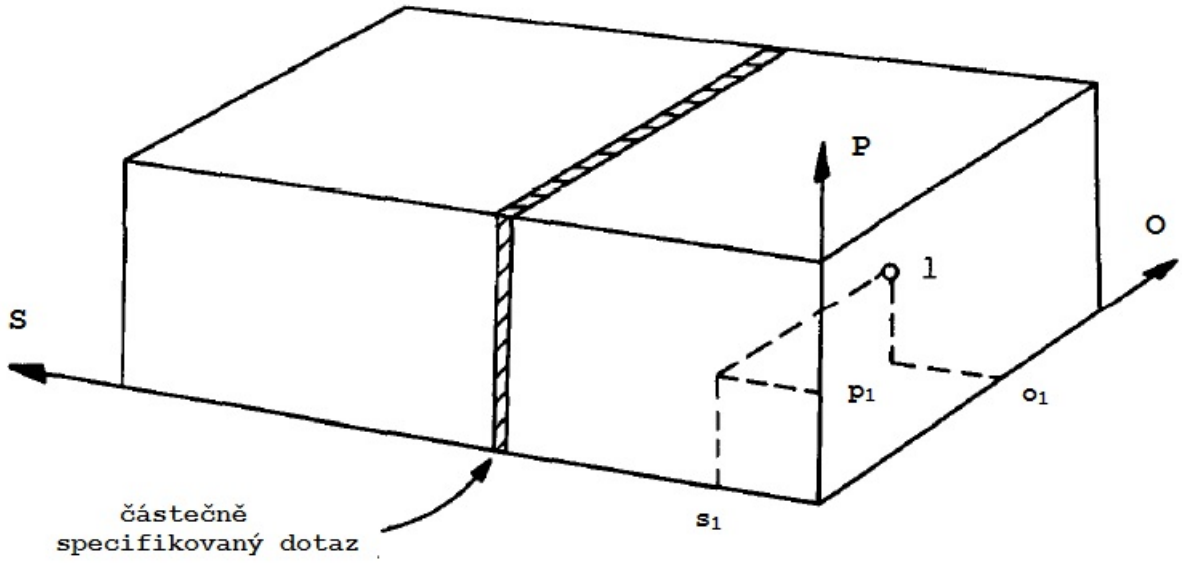
Jedná se tedy o adaptivní, symetrickou a více klíčovou datovou strukturu, která byla původně navržena, aby zvládala přistupovat k úložišti, které se nacházelo na nějakém pevném disku či médiu, které mělo pevně danou velikost. Symetrická proto, že se s každým klíčem v poli určité dimenze se jedná jako s primárním klíčem. Adaptivní pak znamená, že se datová struktura adaptuje její hrany automaticky v závislosti na jejím obsahu, který v sobě musí udržovat.

Dá se o ní mluvit i jako o vícerozměrné struktuře, o čem se taky budeme dále zabývat. Z tohoto důvodu je více indexová neboli více klíčová, když budeme brát klíče jako způsob přístup k daným datům.

Na obrázku 7 pak vidíme, jak můžeme přistupovat k datům ve vícerozměrné struktuře a to tím, že máme tři dimenze (S, P, O) a v nich klíče s_i, p_i, a o_i, dle kterých se dostaneme na určitou pozici a tedy k daným datům, nebo k ukazateli na místo v paměti, kde jsou hledaná data k dispozici.

4.1.2 Struktura

I Grid File má svoji strukturu, která je specifická svými komponenty. Jelikož se tahle struktura se může prezentovat jako n rozměrná s více klíčovým přístupem. Přestavme si, že máme třírozměrnou strukturu a pak bychom potřebovali 3 indexy k určení polohy daného záznamu.



Obrázek 7: Tří rozměrný bitmapa, kde „1“ přestavuje prezenci záznamu s hodnotou determinovanou její pozicí v mapě a „0“ absenci. [15]

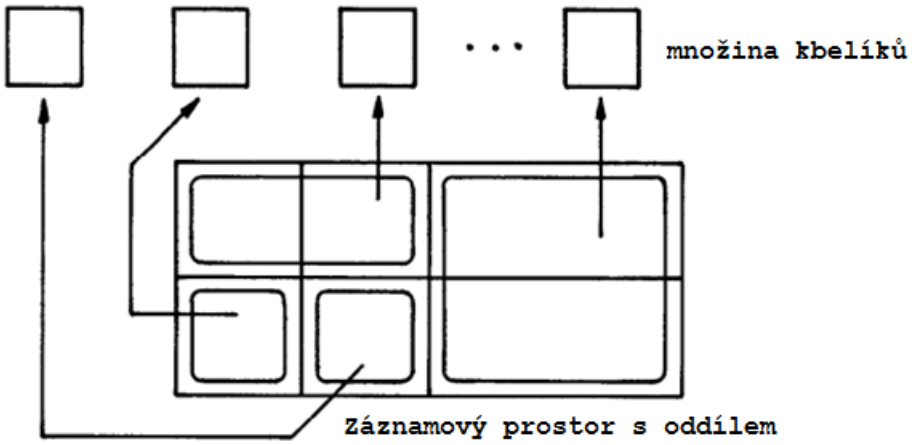
Do češtiny by se tato struktura dala přeložit jako vícerozměrná mřížka, která se skládá:

1. z adresáře (angl. grid directory) – řídící datová struktura, která podporuje operace potřebné k obnovení konvexních přiřazení definované nad tím, když kbelíky přetečou, což jednoduše znamená, že dojde k přerozdělení regionů a tudíž i mřížky, a proto se změní i adresář
2. z množiny regionů, kde každý region odkazuje na daný kbelík, ve kterém se nacházejí odpovídající data
3. a množiny kbelíků – datový blok (angl. bucket), který obsahuje data

Adresář se ještě dělí na (k je celé číslo):

1. dynamické k-rozměrné pole nazvané mřížka (angl. grid array) – jeho elementy (ukazatelé na data sektory neboli regiony) jsou jeden ku jedné korespondující s bloky v oddílu.
2. a k jedno-rozměrných polí nazvané Stupnice (angl.. linear scales) – každá stupnice definuje oddíl dané dimenze.

Na obrázku 8 můžeme vidět mřížku a v ní dané regiony, jenž každý obsahuje odkaz na nějaký kbelík, do něhož můžeme přistoupit či provést nějakou operaci. [15]



Obrázek 8: Konvexní přiřazení datových bloků ke kbelíkům. [15]

4.1.3 Implementování datové struktury

V této podkapitole si ukážeme jak implementovat Grid File, do kterého budeme vkládat trojice, a tudíž bude muset ty data v sobě nějakým způsobem udržovat. Implementace bude probíhat v C++.

Nejdříve si budeme muset vytvořit základní strukturu. V našem případě se bude jednat o trojrozměrnou datovou strukturu, kterou si budeme moci představit jako kvádr, a jelikož budeme pracovat trojicemi, neboli s daty obsahující tři čísla, proto se budeme zabývat třemi dimenzemi. Pak bych to mohl naimplementovat jako trojrozměrné pole:

Region **** gridFileStrukture;

Region představuje datový typ, jenž si můžeme představit jako oblast v tom „kvádru“, do které budeme přistupovat a vkládat do ní data a v závislosti na počtu dat se bude daný region měnit (dělit se na menší celky). Hvězdičky znamenají ukazatelé na další dimenzi, neboli pole, Jednoduše řečeno se jedná o pole polí, která ukazují na další pole a ty ukazují už na zmiňované regiony. Tři hvězdičky poukazují na tři dimenze, ale ta čtvrtá už je jen ukazatel na region.

Dále si musíme nadefinovat region. Nejedná se o nic jiného než o nějakou třídu, která v sobě bude mít nějaké vlastnosti a chování. Laicky řečeno bude obsahovat 3 rozsahy, pro každou dimenzi jednu, pomocí kterých určíme dané trojice, které se budou v daném regionu sdružovat, např.:

rozsah S od 0 – 500 - říká, že tu mohou být trojice, kde S náleží danému rozsahu

Každý region nadále obsahuje kbelík, jenž je definován jako nějaké sekvenční pole, které obsahuje trojice, a když toto pole se zaplní, dojde k rozdělení regionu na dva menší a dané trojice s kbelíku se přerozdělí dle rozsahů do starého a nového kbelíku.

Pak si už jen vytvořit stupnice pro každou dimenzi jednu a ty se pak budou v závislosti na zmenšování regionu měnit, neboli upravovat své rozsahy. Budou představovány jako pole, kde každá hodnota bude představovat hranici rozsahu např.: stupnice O s hodnotami 0, 100, 500, Max (10000).

4.1.4 Přístup k záznamu

V této ukázce si ukážeme jak přistoupit k samotným datům ve 3D mřížce.

Nadefinujeme si adresář G pro trojrozměrný prostor, který je charakterizovaný:

1. přirozená čísla nx > 0, ny > 0 a nz > 0 (rozsah adresáře)
2. přirozená čísla $0 \leq cx < nx$, $0 \leq cy < ny$, $0 \leq cz < nz$ (současný element adresáře a současný blok)

Adresář se skládá z:

1. troj-rozměrného pole G(0 . . . , nx - 1, 0 . . . , ny - 1, 0 . . . , nz - 1) („mřížka“) a
2. jedno-rozměrného pole X(0. . . , nx), Y(0 . . . , ny), Z(0 . . . , nz) („lineární stupnice“)

Potom operace přístupu definovaná nad adresářem může vypadat:

přímý přístup G(cx, cy, cz)

Pak tedy předpokládejme záznamový prostor s atributy „podmět“ s doménou 0 . . . 1000, atributy „vlastnost“ 0 . . . 100 a atributy „předmět“ 0 . . . 1000. Mějme:

X = (0, 250, 500, 750, 1000); Y = (0, 25, 50, 75, 100) Z = (0, 250, 500, 750, 1000)

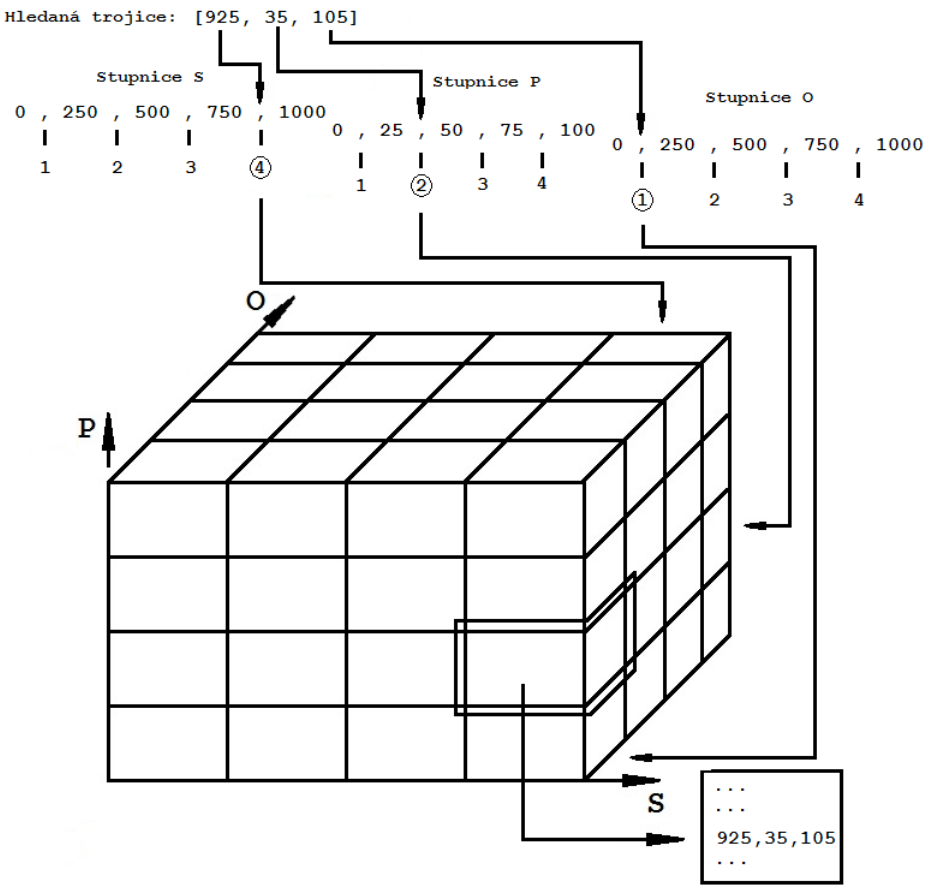
Operace FIND pro plně specifikovaný dotaz (r1, r2, . . .) jako například: FIND(925, 35, 105) je proveden a ukázán na obrázku 9.

Hledaná hodnota 925 je převedena do intervalového indexu 4 skrze hledání na stupnici X a 35 je převedena do indexu 2 na stupnici Y a 105 je převeden do indexu 1 na stupnici Z. Potom indexy 4, 2 a 1 poskytují přímý přístup do daného regionu, ve kterém se může nacházet daná trojice.

4.1.5 Dynamičnost Grid File

Dynamičnost se v tomto případě myslí rozdělení regionu, které může nastat při přetečení maximální hodnoty v kbelíku, který je přiřazen k danému regionu anebo sloučení regionu, jenž nastane při smazání dat.

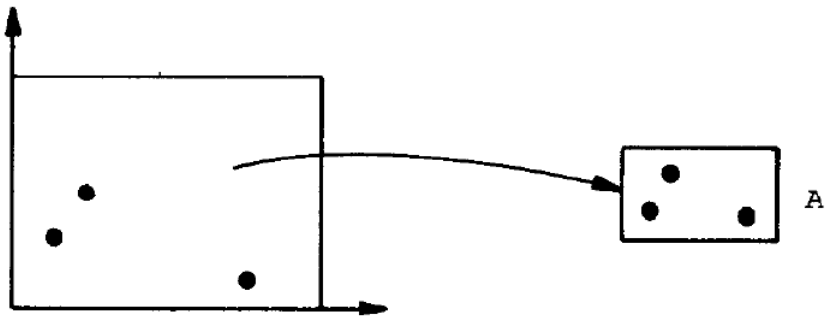
Dynamické chování je nejlépe vysvětleno ukázáním příkladu, kde si vezmeme nějaký region a budeme do něj vkládat data. Pro jednoduchost si to ukážeme na dvojrozměrném poli a místo



Obrázek 9: Přímý přístup k získání jedné trojice.

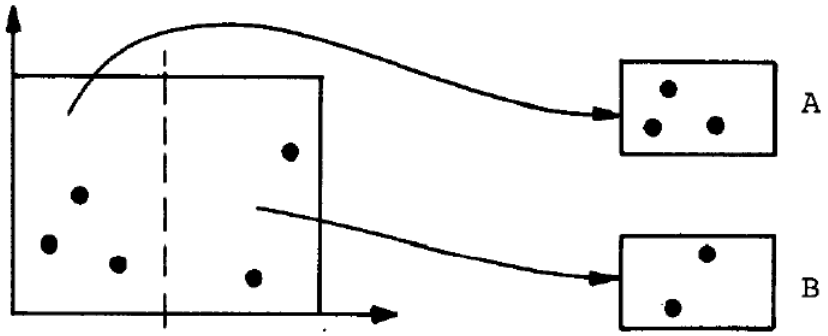
adresáře, kde elementy odpovídají jedna ku jedné s bloky, budeme kreslit ukazatele na kbelíky vznikající přímo v blocích.

Nejprve si nadefinujeme kbelík A, v našem případě kapacity c = 3, jenž nám představuje celý záznamový prostor.



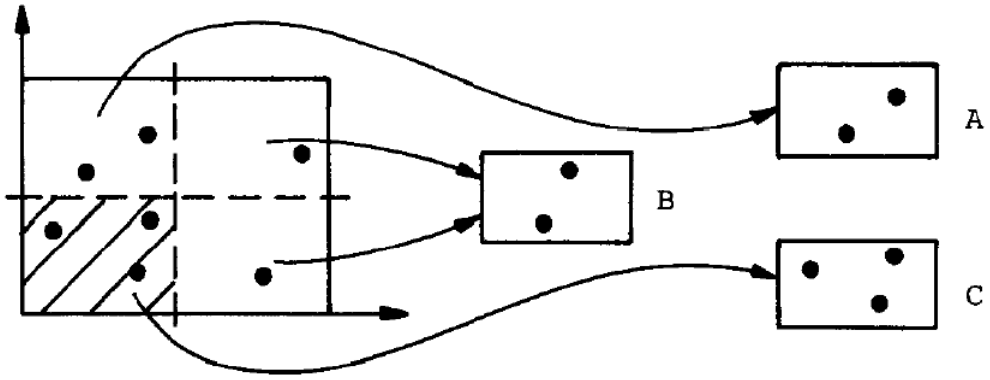
Obrázek 10: Kbelík před rozdělením [15]

Když kbelík A přeteče, záznamový prostor je rozdělen a nový kbelík B je vytvořen a záznamy, které leží na jedné straně prostoru (regionu) jsou přemístěny ze starého kbelíku do nového.



Obrázek 11: Rozdělení prostoru [15]

Jestli kbelík A znovu přeteče, respektive region, ve kterém se nachází (levá polovina prostoru) je rozdělen podle nějakého pravidla rozdělení. Nejjednodušší pravidlo pro rozdělení je v jeho polovině, ale pokud, jak vidíme černé tečky na obrázku 3.5, se budou všechny nacházet u sebe v nějakém malém místě, tak se pomyslná půlící čára povede mezi nimi, aby došlo k jejím rozumnému přerozdělení.



Obrázek 12: Opětovné přerozdělení a dva bloky ukazující na stejný kbelík [15]

Pak když se přidají další záznamy do kbelíku A, dojde k dalšímu rozdělení a záznamy ležící v levém dolním bloku budou přemístěny do nového kbelíku C, ale jak si můžeme všimnout, pravý blok se sice mohl též rozdělit, ale oba bloky (horní i dolní) ukazují na stejný kbelík B. Dále se to rozděluje dle podobného či stejného pravidla. [15]

4.2 AVL-strom

Než se něco povíme o AVL-stromech, tak musíme zmínit, že se jedná v základě o binární vyhledávací strom, jenž jednoduše znamená, že každý uzel stromu obsahuje daný klíč, dle kterého se určuje, zda se například při vyhledávání podíváme do levého uzlu (potomek s menší hodnotou

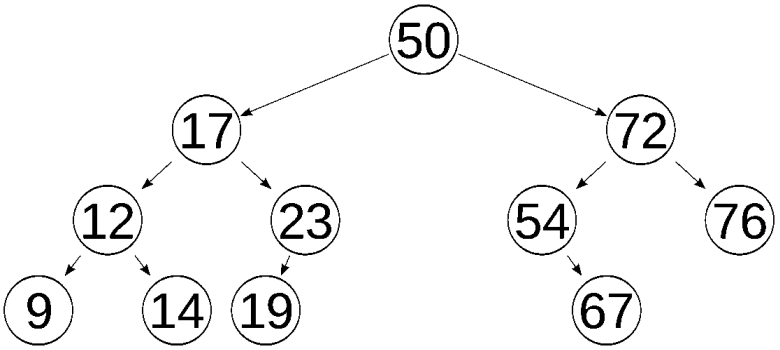
klíče) anebo do pravého uzlu (potomek s větší hodnotou klíče), a proto se jedna o binární vyhledávací strom, máme jen dvě možnosti, kam dále postupovat (pokud samozřejmě nepočítám možnost, že se klíče rovnají).

Největší rozdíl v čem se AVL-stromy liší od binárních stromech je to, že se jedná o vyvážené vyhledávací stromy, kde rozdíl výšek jednotlivých podstromů může být nejvýše 1. Jednoduše řečeno, v každém podstromu najdeme podobný počet potomků a tedy nemůže dojít, že by v nějakém podstromě bylo o mnohem (například o 10) více uzlů než v jiných. Když se třeba při vkládání nových uzlů tato podmínka poruší, dojde k vyrovnání stromu.

Název AVL-strom vznikl tak, že byl pojmenován po dvou ruských matematicích Georgii Adelson-Velsky a Evgenii Mikhailovich Landis, jenž se shodují dle prvního písmena jejich příjmeních a poprvé byl publikován v roce 1962 v článku „An algorithm for the organization of information“.

AVL-stromy se i vyznačují tím, že mají časovou složitost O (log n) při vykonávání základních operací jako jsou:

1. vyhledávání uzlu dle klíče
2. vložit nový uzel
3. smazat uzel s určitým klíčem



Obrázek 13: Příklad AVL-stromu [16]

4.2.1 Konstrukce AVL-stromu

Při konstrukci AVL-stromu si musíme uvědomit, že se jedná o vyváženou strukturu, a proto ji musíme k tomu i přizpůsobit. Problém nastává při tom, když je nový uzel přidán do stromu, pak se musíme přemísťovat zpět po předešlé cestě všech předků nového uzlu a testovat jednu ze tří podmínek:

1. už jsme dosáhli kořene stromu, a proto nemusíme použít proceduru na vyvážení stromu

2. narazili jsme na předka, jehož kratší podstrom byl prodloužen přidáním nového uzlu a tím se taky zruší procedura na vyvážení stromu, protože nový uzel nenarušil podmínku vyváženosti, jen se stalo to, že kratší podstrom byl prodloužen na stejnou délku jako ten sousední v rámci stejné výšky stromu

3. narazili jsme na předka, jehož delší podstrom byl prodloužen přidáním nového uzlu a tím pádem ten delší podstrom překračuje ten kratší podstrom o větší množství uzlů, než je povoleno, tím pádem se porušila i výška stromu v jednotlivých podstromech a tím se spustí procedura na vyvážení stromu [17]

4.2.2 Vyvážení výšky stromu

AVL-stromy jsou stavěny podle výškově vyvažovacích algoritmů. AVL-strom je binární vyhledávací strom, jestli výšky levých potomků a pravých potomků jakýkoliv uzlů ve stromě se neliší o více než 1. Hodnota vyváženosti toto indikuje dvěma bity:

1. +1: vyšší pravý potomek

2. 0: stejné výšky

3. -1: vyšší levý potomek

Převrácený algoritmus pro konstruování AVL-stromu se skládá ze tří fází:

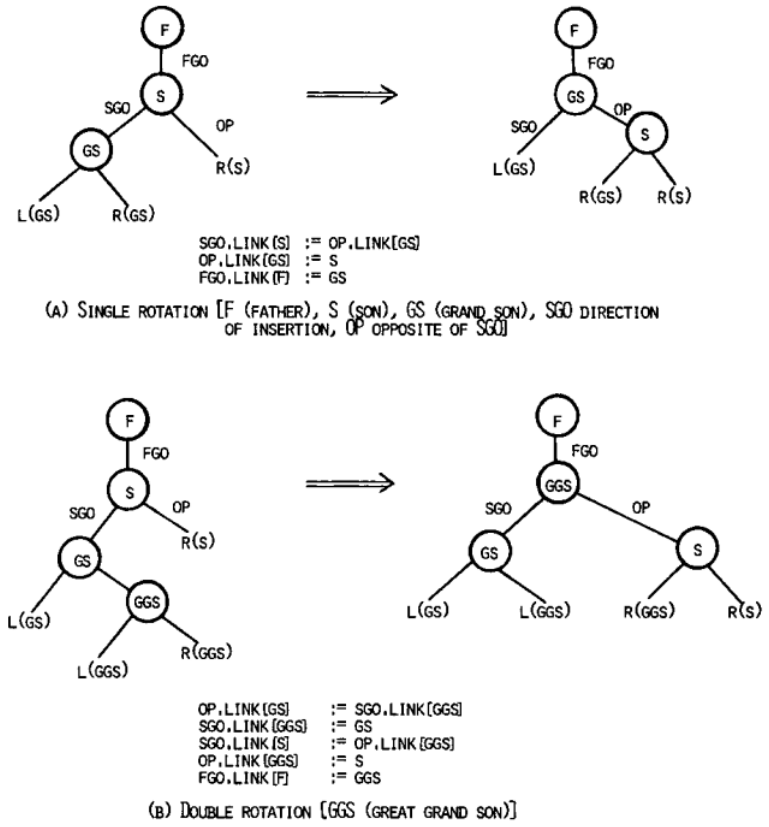
1. Najdi místo vložení a kritický uzel.

2. Modifikuj hodnotou vyváženosti mezi kritickým uzlem a novým listovým uzlem.

3. Vyvaž kritický uzel, pokud je to nutné, jinak změn jeho hodnotou vyváženosti.

Kritický uzel nalezený ve fázi 1 je poslední uzel na cestě vložení nového uzlu nebo kořene, jestli žádný takový uzel neexistuje. Proto fáze 1 je podobná ke konstrukci náhodného binárního stromu s kontrolou přidáním kritického uzlu. Fáze 2 zahrnuje převrácený algoritmus. Modifikace vyváženosti je okamžitá a závisí na směru cesty vložení. Ve fázi 3 vyvážení nastane, jestli podstrom, jehož kořen je kritický uzel, se stane více nevyváženým, tj. jestli směr cesty vložení a přítomnost hodnoty vyváženosti splývá.

V tomhle případě, volání kritického uzlu S na obrázku 15, jestli S a jeho potomek GS mají klíče oba velké (nebo malé) než nový listový uzel, tj. jestli SGO = GSGO, pak máme jednoduchou rotaci, jinak nastane dvojitá rotace. Jestli podstrom se stane méně nevyváženým, modifikace hodnoty vyváženosti kritického uzlu je dostatečná. Konečně jestli kořen byl kritickým uzel s vyvážeností 0, jeho hodnota vyváženosti musí být změněna. [18]



Obrázek 14: Rotace [18]

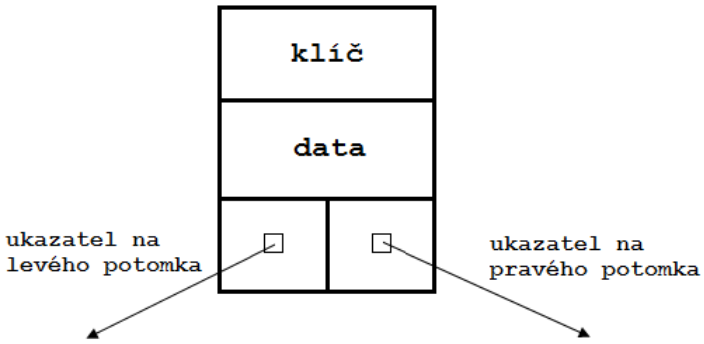
4.2.3 Implementování datové struktury

Když budeme implementovat AVL-strom, nesmíme zapomenout, že se jedná o stromovou strukturu a to znamená, že budeme muset vytvořit nejprve strukturu představující strom samotný a pak samozřejmě nesmíme zapomenout na uzel, kde budeme ukládat naše data, což v našem případě jsou trojice.

Strom si nadefinujeme jako nějakou třídu, ve které budeme definovat příslušné metody. Než se vydáme na metody vložení či vyhledávání, tak si musíme na definovat důležité metody pro vyvážení stromu, tyhle metody, dalo by se i říct, představují smysl AVL-stromu, ale nejprve si nadefinujeme uzel. Jelikož budeme pracovat s trojicemi, tak můj uzel by mohl vypadat asi takhle:

Uzel obsahuje:

1. tři klíče představující trojici s, p, o a vždy podle jednoho z tohoto klíče se budeme řídit,
2. ukazatele na pravého a levého potomka
3. a hodnotu výšky uzlu



Obrázek 15: Uzel AVL-stromu

Pak bychom si jako první mohli nadefinovat metodu, která nám vrátí výšku daného uzlu, kterou budeme potřebovat pro další operace:

vratVyšku - parametr bychom do ní dali uzel a návratový typ by byl celé číslo

Následné si musíme vytvořit něco, co nám bude vracet rozdíl výšek pravého a levého potomka, můžeme si ji nazvat:

bFactor – parametr bude mít opět uzel a vracet bude už zmiňovaný rozdíl výšek

Samozřejmě budeme muset i spravovat výšku a to budeme dělat pomocí metody:

spravVýšku – parametr je uzel, kde se bude měnit výška a její účelem je porovnat výšky pravého a levého potomka a ta, která je větší, tak ji zvětšit o 1 a nastavit do toho daného uzlu (otec těch potomků).

Ovšem nesmíme zapomenout ani na provedení levé a pravé rotace. Ukážeme si například pravou rotaci a ta by mohla vypadat asi takto:

1. máme uzel n
2. do pomocné proměnné si uložíme ukazatel na levého potomka
3. do ukazatele levého potomka vložíme ukazatele na pravého potomka
4. do levého ukazatele vložíme samotný uzel n
5. spravíme výšku uzlu n, pak i uzlu v pomocné proměnné a ten následné vrátíme
6. opakujeme, dokud dané výšky neporušují pravidlo vyváženosti

Potom levý rotace je jen obracená kopie té pravé.

Jako další důležitou operaci si můžeme na definovat metodu vyvážení stromu a nazveme si ji třeba: provedVyvážení a její algoritmus by mohl vypadat takto:

1. máme uzel n
2. nejprve sprav výšku uzlu n
3. pak zjisti bFactor uzlu n a jestli se rovná 2 a následně bFactor jeho pravého potomka je menší než 0, pak proveď pravou rotaci v pravém potomkovi, jinak proveď levou rotaci v uzlu n a tohle prováděj, dokud nedojde k listovému uzlu
4. pokud zjistí bFactor uzlu n se rovná -2 a následně bFactor jeho levého potomka je větší než 0, proveď levou rotaci v levém potomkovi, jinak proveď rotaci pravou uzlu n a tohle prováděj, dokud nedojdeš na listový uzel
5. pokud bFactor uzlu n není 2 ani -2, tak vrať uzel n

Tento algoritmus je nastaven pro cyklický průchod stromem, proto se vrací uzly, vždy se prochází z kořene do všech podstromů a provádí se vyvážení. A v poslední řadě si vytvoříme metody na vložení, smazání či vyhledávání uzlů ve stromu. A nakonec nesmíme zapomenout vytvořit kořen, který na začátku bude odpovídat prázdnému uzlu, a postupně se k němu budou připojovat nové uzly a tím se bude strom zvětšovat a vyvažovat.

4.2.4 Vyhledávání

Jelikož se jedná o binární vyhledávací strom, tak vyhledávání je jednoduché. Vše začíná u kořene, ten navštívíme jako první a pak už budeme opakovat ten samý algoritmus, kde se budeme porovnávat hledanou hodnotou h s klíčem uzlu k:

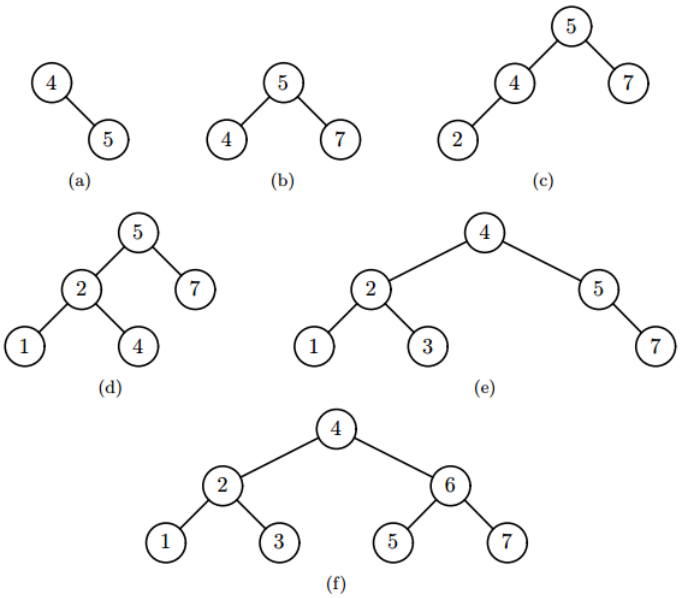
1. h = k – našel se hledaný uzel
2. h > k – pokračujeme do pravého potomka, protože tam mohou být uzly s větší hodnotou klíče
3. h < k – pokračujeme do levého potomka, tam se mohou nacházet uzly s menšími klíči

Pokud dojdeme do situace, že ukazatel na levého nebo pravého potomka je prázdný, což znamená, že se jedná o listový uzel, pak konstatujeme, že hledaný uzel nebyl nalezen.

4.2.5 Vkládání

Vložení nového prvku by se mohlo zdát, že kvůli vyváženosti to bude složité, ale v podstatě se jedná to samé jako u vyhledávání. Začneme v kořeni a budeme opět porovnávat hodnotu klíče, kterou chceme přidat s klíčem uzlu, ve kterém se právě nacházíme a stejného pravidla, pokud nový klíč je větší přejdeme do pravého potomka, zda je menší, přejdeme do levého potomka, a kdyby byly klíče stejné, tak se pozná, že ten klíč už strom obsahuje a novy uzel se nepřidá, pokud samozřejmě nechceme duplicitu klíčů, jinak bychom ho mohli přidat do levého potomka,

to by záleželo na dané situaci, ale v mém případě každý klíč je jedinečný ve stromě. Tohle se opakuje, dokud se nenarazí na listový uzel, kde ukazatel na potomka je prázdný a v tom případě se do jednoho z nich přidá uzel s novým klíčem, tedy dostaneme nový listový uzel. Pak nastává operace vyvážení, která provede jakoby kontrolu stromu a to tím, že projde strom a zkoumá výšky uzlu, zda nalezne v nějakém podstromu více uzlu, neboli rozdíl výšek je větší než 1, tak provede vyvážení stromu.



Obrázek 16: Vkládání do AVL-stromu [19]

4.2.6 Mazání

U mazání uzlu jsou dva případy, pokud se maže listový uzel, tak v podstatě není co řešit, jen se opět dle algoritmu porovnávaní klíčů zjistí, zda jít do pravého či levého potomka dokud se klíče neshodují, to znamená, že se našel uzel, co se má smazat, pokud je tedy listový, smaže se a popřípadě se provede vyvážení, pokud je to nutné. Druhý případ je ten, že se bude mazat uzel, který není listový a má tedy své potomky, pak se zkontroluje ukazatel na pravého potomka, jestli je prázdný a tím pádem, nemá žádný pravý podstrom, pak se ten uzel smaže a na jeho místo se vloží ukazatel na levý podstrom, ale pokud obsahuje pravý podstrom, pak se bude muset najít v tom pravém podstromu nejmenší uzel, smazat ten daný uzel a nahradit ho tím nejmenším uzlem a jako jeho levý podstrom vložit levý podstrom toho smazaného uzlu a následné provést vyvážení, které upraví strom do správné podoby.

5 Testování

Tahle část se zabývá testováním datových struktur a to především našich naimplementovaných struktur: AVL-strom a Grid File. Dále se budou testovat již přichystané a odladěné struktury, ze kterých testujeme: R-strom a B-strom. Všechno testování probíhá ve frameworku, který nám byl poskytnut vedoucím mé bakalářské práce. Framework obsahuje určité datové struktury, které jsou již implementované a čekají jen na testování. Naším úkolem bylo jen upravit Framework tak, že tam přidáme své implementované datové struktury, které otestujeme spolu s danými strukturami ve frameworku.

Testování probíhá tím způsobem, že každou datovou strukturu budeme dotazovat sadou dotazů, a pak se vyhodnotí čas, za který to daná struktura vykonala. Daný Framework byl vytvořen a prezentován ve vývojovém prostředí Microsoft Visual Studio 2013 a v programovacím jazyce C++.

5.1 Použitý stroj na testování

Na provedení daného testování jsme použili náš notebook, pomocí kterého jsme implementovali své struktury, a rovněž jsme provedli dané dotazování v rámci našeho testu, Dalo by se říct, že i trochu záleží, na jakém stroji to testování probíhá, kdybychom měli jiný stroj, například s horšími parametry, mohli by časy daných dotazů vypadat o trochu jinak.

Tabulka 3: Parametry použitého notebooku

Model	Lenovo Y50-70 Touch
Operační systém	Windows 7 Professional
Procesor	Intel Core i7-4710HQ 2,5Ghz
Operační paměť	16GB DDR3 1600 MHz
Pevný disk	1TB 5400 ot/min

5.2 Rozsahový dotaz

Hlavní test, o kterém tahle bakalářská práce je, se provádí způsobem tak, že v daném frameworku je nachystaná sada dotazů, které mají otestovat struktury a naměřit čas, za který to zvládnou provést. Samotný dotaz se pak obsahuje, co je pro nás důležité, dvě věci:

1. spodní mez – trojice označující spodní hranici rozsahu trojic
2. horní mez – trojice označující horní hranici rozsahu trojic

Když si vezmeme dotaz, který má spodní mez: 54 2 0 a horní mez: 54 2 MAX, kde MAX odkazuje maximální hodnotě datového typu integer (2^{32}), pak tímto dotazem říkáme, že hledáme trojice (S P O), kde podmět S odpovídá hodnotě 54 s vlastností P s hodnotou 2, jehož předměty O jsou v rozsahu 0 až 2^{32}.

5.3 Testování dotazů

Samotné dotazování pak probíhá tak, že každá struktura má svoji metodu, která se pravděpodobně jmenuje „RangeQuery“ a jejími hlavními parametry jsou už zmiňované meze, ve kterých vyhledává trojice. To, co nás ještě může zajímat mimo času, který se při tom vyhledávání měří a následné vypíše, tak je i počet nalezených trojic. Provedli jsme testování na čtyřech strukturách a v tabulce 4 pak můžeme vidět dané časy jednotlivých dotazů pro každou strukturu. Jednotlivé dotazy byly provedeny deset tisíckrát, aby bylo co měřit.

Tabulka 4: Naměřené časy jednotlivých datových struktur

		Datové struktury			
Číslo dotazu	**Počet trojic**	**B-strom[s]**	**R-strom[s]**	**AVL-strom[s]**	**Grid File[s]**
1	5	0,057	0,94	0,99	46,46
2	2	0,63	0,857	0,89	46,08
3	3	0,036	1,13	2,01	48,52
4	5	0,043	0,928	1,05	48,54
5	1	0,036	1,521	0,99	49,55
6	1	0,036	2,153	2,003	50,524
7	6	0,049	9,09	109,95	50,54
8	2	0,036	2,297	1,03	67,252
9	1006	1,533	221,026	687,45	679,542
10	1006	1,509	364,933	238,21	830,985
11	32	0,006	241,162	8,05	644,245
12	4	0,049	9,567	247,54	466,542
13	4496	6,26	599,364	1095,36	1801,254
14	5388	0,062	607,039	1277,32	2065,248
15	149	0,265	38,444	491,36	559,352
16	1265	2,054	186,539	432,25	878,325
17	56	0,119	343,565	14,8	565,365
18	1437	2,252	149,244	652,36	1027,352
19	59420	93,904	317,033	3825,36	12151,325
20	16134	26,327	386	1321,25	2298,325
21	16134	25,05	718,411	512,365	2660,324
22	160292	253,607	1068,43	6983,254	25543,25
23	216024	332,898	774,198	12865,365	33752,358
24	18869	28,641	397,667	1175,365	3752,227
25	1006	1,55	5,833	66,35	351,35
26	14	0,056	9,927	1,89	47,256
27	13	0,056	10,081	2	46,254
28	13	0,051	10,25	1,94	48,256
29	378212	594,835	1015,65	18267,64	70546,37
30	83694	129,472	682,382	15605,3	27312,325

Jak jsme si mohli všimnout, naměřené časy se, jak bychom řekli, se celkem rapidně liší. Jak vidíme v tabulce 4, B-strom si vedl nejlépe a hned za ním je R-strom, který sice má horší časy, ale

stále lepší než naše implementované stukrtury. Naše struktury si vedly mnohem hůře, než jsme zamýšleli. Ale takový rozdíl by mohl být způsobena tím, že první dvě struktury jsou odladěné a nejedná se o základní typ B-stromu, ale jejich určité varianty, to pak může znamenat, že konstrukce je vylepšená tak, že průběh vyhledávání je pak zcela odlišný od způsobu vyhledávání v našich strukturách. Naše stromová struktura byla implementována jako základní AVL-strom bez žádných vylepšení a chtěli jsme ji porovnat s odladěnými strukturami a naše očekávání to splnilo, byla pomalejší. Druhá struktura Grid File je pak celkově nejpomalejší, což jsme i očekávali, nejedná se o stromovou struktura, ale o více rozměrnou datovou strukturu a v ní se pak používá sekvenční průchod jak dimenzemi, tak i poli, kde se nacházejí samotné trojice, a čím více dat, tím se mřížka v Grid Filu více rozděluje a vyhledávání se komplikuje, a proto její výkon není zrovna nejideálnější.

Dále jsme si mohli všimnout toho, že čím více nalezených trojic, tím se čas zvětšuje.

5.4 Velikosti datových struktur

V tabulce 5 si můžeme prohlédnout velikosti jednotlivých datových struktur. Dle velikosti můžeme vidět, jaká struktura zabírá nejvíce místa v paměti v závislosti na stejném počtu dat. U datových struktur typu strom, si velikost vypočítáme tak, že počet uzlů vynásobíme velikostí jednoho uzlu a u Grid Filu velikost vypočteme tím, že vynásobíme velikost regionu s jejím počtem.

Tabulka 5: Velikosti datových struktur

Datová struktura	**Celková velikost [MB]**
B-strom	77,901
R-strom	25,4
AVL-strom	48,541
Grid File	660,259

5.5 Doba vytvoření datových struktur

Doba vytvoření nám říká, za jakou dobu se struktura na inicializuje a bude připravena k použití. Dle tabulky 6 si můžeme všimnout, že R-strom potřebuje mnohem více času na vytvoření než ostatní struktury.

Tabulka 6: Časy potřebné na vytvoření struktur

Datová struktura	**Čas [s]**
B-strom	11,323
R-strom	50,209
AVL-strom	6,405
Grid File	22,044

5.6 Počet přístupu

Počtem přístupu se myslí, kolikrát určitá struktura provede přístup do své struktury a provede nějakou operaci například: procházení uzlu nebo pole s daty. V tabulce 7 můžeme vidět, kolik bylo potřeba přístupu pro provedení jednotlivých dotazů. Dále bychom upozornili, že počet přístupu souvisí i s počtem nalezení trojic. Kdybychom to chtěli upřesnit, pokud nebudeme počítat B-strom, jehož počet přístupu se shoduje s jeho výškou, a tedy v našem případě je jeho počet přístupu stejný pro všechny dotazy, ale pro ostatní měřené struktury si můžeme všimnout, že čím více trojic bylo nalezeno, tím větší je i větší počet přístupu.

Tabulka 7: Počet přístupu při provádění dotazů

	Datové struktury			
Číslo dotazu	**B-strom**	**R-strom**	**AVL-strom**	**Grid File**
1	4	13	85	3
2	4	11	187	3
3	4	16	60	3
4	4	14	80	3
5	4	20	80	2
6	4	30	80	2
7	4	120	3228	5
8	4	30	88	3
9	4	1321	13452	61
10	4	2183	3049	63
11	4	1397	112	11
12	4	122	6482	5
13	4	4446	18011	1133
14	4	4487	21570	239
15	4	309	16693	150
16	4	1105	17806	213
17	4	2014	187	53
18	4	954	131785	216
19	4	1580	375164	275
20	4	2198	133678	286
21	4	5132	64556	289
22	4	6454	641192	2005
23	4	3804	864114	358
24	4	2294	75485	270
25	4	1630	4352	61
26	4	123	61	8
27	4	127	58	8
28	4	129	68	9
29	4	4276	1755386	1141
30	4	4388	334795	518

5.7 Zdroj dat

Jako zdroj dat jsme použili databázi dat, kde jsou uloženy všechny trojice, se kterými v testování pracuji. Konkrétně jsme použili soubor „lubm8.nt“, kde se fyzicky nacházejí všechny trojice, nad kterými se provádí testování. Počet trojic obsažených v souboru je cca 1 430 000.

Jedná se o syntetická (umělá) data, vygenerována data generátorem, který vytvoří surová RDF data, následně upravíme vygenerované data tak, že je nadstavíme dalšími daty pomocí jazyka OWL, pak budeme moci odvozovat další data, například při vyhledávání nějakého studenta nebudeme vědět pouze to, že je to student, ale budeme i moci odvodit, že se jedná o osobu nebo můžeme také určit, že je to člověk, potom z takových dat můžeme dostat skutečnější výsledky, než kdybychom testovali umělá data. [20]

6 Závěr

V této bakalářské práci byly naimplementované dvě datové struktury: AVL-strom a Grid File a následně byly naplněny vhodnými daty a otestovány dotazy.

Dle hodnot, které jsme naměřili, konstatujeme, že B-strom, a v závěsu za ním R-strom, jsou datové struktury k ukládání i vyhledávání dat nejvhodnější z těch, které jsme testovali. Naše struktury bychom úplně nezavrhnuli, ale jelikož se nejedná o jejich vylepšené varianty, ale pouze o základní datové struktury, tak jejich časy se nemohou vyrovnat s těmi dvěma předchozími.

AVL-strom není špatná struktura pro zpracovávání dat, jako jsou trojice. Věřím, že kdyby byl AVL-strom upraven na nějakou z jeho vylepšených variant, časy by byly lepší, ale my se zaměřili pouze na jednoduchý a hlavně základní AVL-strom a vyzkoušet si, jak obstojí vůči lépe optimalizovaným strukturám, časy to jen dokazují, že když chceme použít nějakou stromovou strukturu, měli bychom volit jednu z „vylepšených“ variant stromů, kde například: procházení samotných trojic, budeme řešit optimalizovanějším způsobem.

S Grid Filem jako strukturou pro zpracování trojic bych podle našeho názoru řekli, že se pro tohle problematiku nehodí. Nejedná se o špatnou strukturu, dá se s ní dobře pracovat, ale i přesto, má pár svých nedostatků jako je například to, že při větším počtu dat stále více rozděluje do menších bloků a může se stát, že v určitých situacích může nastat to, že vznikne mnoho bloků, ve kterých budou kbelíky zaplněny velmi malým počtem trojic, nebo budou dokonce prázdné, ale to se dá optimalizovat tak, že prázdné bloky nebudou mít kbelíky a tím by zabírali místo v paměti, ale i přes toto opatření dochází k inicializaci spoustu místa, a proto si dovolujeme říct, že tato struktura zabírá poměrně hodně místa v paměti. Nejen kvůli naměřeným hodnotám, ale i skutečnosti, že mnoho dnešních databází používá jako svou datovou strukturu právě B-strom a po případě i další, jak se můžeme vidět v [4], se domníváme, že stromové struktury budou vhodnější než náš naimplementovaný Grid File.

Díky této bakalářské práci jsme si mohli vyzkoušet, jak implementovat stromovou strukturu. Následně jsme se poprvé setkali s vícerozměrnou strukturou jako je Grid File, která není moc známá a moc se o ní nemluví. Nejzajímavější bylo, tu strukturu vůbec pochopit, jak s ní pracovat a jak vůbec do ní ty data uložit. Nejhorší pro nás bylo její fakt, že se stále rozděluje do dalších regionů, a tím je více složitější. Dle naměřených hodnot si troufáme říct, že se nejedná o vhodnou strukturu pro ukládání trojic. S AVL-stromem už nebyla taková potíž a její výsledky nejsou až tak hrozné, když vezmeme v úvahu, že jeho varianty by byly mnohem rychlejší.

Literatura

[1] PALMER, Sean B. The Semantic Web: An Introduction [online]. Dostupné z: http://infomesh.net/2001/swintro/

[2] Berners-Lee, Tim. Semantic Web Language stack [online]. Dostupné z: https://www.w3.org/2009/Talks/0120-campus-party-tbl/#(14)

[3] GROPPE, Sven. Data Management and Query Processing in Semantic Web Databases. 1. Springer-Verlag Berlin Heidelberg, 2011. ISBN 978-3-642-19357-6.

[4] David Célestin Faye, Olivier Curé, and Guillaume Blin. "A survey of RDF storage approaches". In: ARIMA Journal 15 (2012). Dostupné z: http://arima.inria.fr/015/015002.html

[5] Roman Meca, Michal KRÁTKÝ, Peter CHOVANEC a Filip KŘIŽKA. Data Structures for Indexing Triple Table [online]. Dostupné z: http://ceur-ws.org/Vol-1343/paper19.pdf

[6] Daconta, M. C., Obrst, L. J., Smith, K. T. What Is the Resource Description Framework?, 2007 Dostupné z: http://www.devx.com/semantic/Article/34816

[7] Pendleton, Bryan. A Short History of the BTree [online], 2011. Dostupné z: https://www.perforce.com/blog/110928/short-history-btree

[8] COMER„ Douglas. Ubiquitous B-tree. ACM Computing Surveys (CSUR) [online]. ACM, 1979, (2), 121-137. Dostupné z: http://people.cs.aau.dk/~simas/aalg06/UbiquitBtree.pdf

[9] Data Structures and Algorithms. B-Trees [online]. Dostupné z: http://lcm.csa.iisc.ernet.in/dsa/node122.html

[10] Bayer, R.; McCreight, E. (1972), "Organization and Maintenance of Large Ordered Indexes", Acta Informatica 1 (3): 173–189, doi:10.1007/bf00288683 Dostupné z: http://www.minet.uni-jena.de/dbis/lehre/ws2005/dbs1/Bayer_hist.pdf

[11] Guttman, Antonin. R-trees: a dynamic index structure for spatial searching [online]. Vol. 14. No. 2. ACM, 1984. Dostupné z: http://pages.cs.wisc.edu/~cs764-1/rtree.pdf

[12] James D. Foley (1995). Computer Graphics: Principles and Practice. Addison-Wesley Professional. p. 13. ISBN 0-201-84840-6. "The term bitmap, strictly speaking, applies only to 1-bit-per-pixel bilevel systems; for multiple-bit-per-pixel systems, we use the more general term pixmap (short for pixel map)."

[13] A. POOLET, Michelle. What's a Bitmap Index? [online]. In: . 2007 [cit. 2016-04-13]. Dostupné z: http://sqlmag.com/business-intelligence/what-s-bitmap-index

[14] O'Neil, Elizabeth, Patrick O'Neil, and Kesheng Wu. Bitmap index design choices and their performance implications [online]. Database Engineering and Applications Symposium, 2007. IDEAS 2007. 11th International. IEEE, 2007. Dostupné z: http://crd-legacy.lbl.gov/~kewu/ps/LBNL-62756.pdf

[15] Nievergelt, Jürg, Hans Hinterberger, and Kenneth C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. ACM Transactions on Database Systems (TODS) 9.1 (1984): 38-71. Dostupné z: http://www.cs.ucr.edu/~tsotras/cs236/W15/grid-file.pdf

[16] Algorithms and Ideas In Java, Self balancing binary search trees comparison [online]. Dostupné z: https://intelligentjava.wordpress.com/tag/scapegoat-tree/

[17] Foster, Caxton C. A generalization of AVL trees. Communications of the ACM 16.8 (1973): 513-517.

[18] Baer, J-L., and B. Schwab. "A comparison of tree-balancing algorithms."Communications of the ACM 20.5 (1977): 322-330.

[19] DVORSKÝ, Jiří. Algoritmy I: Pracovní verze skript [online]. 2007, 157. Dostupné z: http://www.cs.vsb.cz/dvorsky/Download/SkriptaAlgoritmy/Algoritmy.pdf

[20] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. "LUBM: A benchmark for OWL knowledge base systems". In: Web Semantics: Science, Services and Agents on the World Wide Web 3.2 (2005), pp. 158–182.

A Příloha na CD/DVD

Součástí BP/DP je CD/DVD.
Adresářová struktura přiloženého CD/DVD:

- Bakalářská práce
  - Framework
    - Data - složka, která osbahuje soubor s daty
    - Framework - složka, která obsahuje framework v jazyce C++
    - Readme.txt - soubor obsahující dodatečné informace
  - Text - složka obsahuje bakalářkou práci ve formátu pdf
    - Bakalářská_práce.pdf